

Scalability Analysis of Three Monitoring and Information Systems:

MDS2, R-GMA, and Hawkeye

Xuehai Zhang,¹ Jeffrey L. Freschl,² and Jennifer M. Schopf³
hai@cs.uchicago.edu, jfreschl@cs.wisc.edu, jms@mcs.anl.gov

¹*Department of Computer Science, University of Chicago*

²*Department of Computer Science, University of Wisconsin at Madison*

³*Mathematics and Computer Science Division, Argonne National Laboratory*

Keywords: Distributed Systems, Grid Performance Study, Monitoring and Information Systems, Performance Analysis

Abstract

Monitoring and information system (MIS) implementations provide data about available resources and services within a distributed system, or Grid. A comprehensive performance evaluation of an MIS can aid in detecting potential bottlenecks, advise in deployment, and help improve future system development. In this paper, we analyze and compare the performance of three implementations in a quantitative manner: the Globus Toolkit® Monitoring and Discovery Service (MDS2), the European DataGrid Relational Grid Monitoring Architecture (R-GMA), and the Condor project's Hawkeye. We use the NetLogger toolkit to instrument the main service components of each MIS and conduct four sets of experiments to benchmark their scalability with respect to the number of users, the number of resources, and the amount of data collected. Our study provides quantitative measurements comparable across all systems. We also find performance bottlenecks and identify how they relate to the design goals, underlying architectures, and implementation technologies of the corresponding MIS, and we present guidelines for deploying monitoring and information systems in practice.

1. Introduction

A monitoring and information system (MIS) is a key component of a distributed system, or Grid, that provides information about the available resources and their status. An MIS can be used in a variety of ways: a resource broker may query the MIS to locate computing elements for the CPU and memory requirements described by a job, a program may collect a stream of data generated by an MIS to help steer an application, or a system administrator may set the MIS to send a notification when system load or disk space availability changes to identify possible performance anomalies.

In this paper, we provide a quantitative and comprehensive performance analysis of the MIS systems to aid in detecting potential bottlenecks, advise in system deployment, and help improve future development

work. We study the performance of three prevalent MIS implementations: the Globus Toolkit® [20] Monitoring and Discovery Service (MDS2) [10, 37], the European Data Grid [12] Relational Grid Monitoring Architecture (R-GMA) [8, 11], and Hawkeye [23] developed by the Condor project [35]. Each of these systems is in use in production or near-production Grids. We analyze the scalability of the three MIS implementations by running four sets of experiments and examining the effect of a large number of concurrent users, resources, and data sources on the load, throughput, and end-to-end response time of each MIS. To better understand the end-to-end performance, we instrument the main service components of each MIS with NetLogger [21, 53], a real-time performance diagnosis tool, to capture detailed system behavior and to detect performance limitations.

The remainder of this article is as follows. To facilitate the performance evaluation and comparison, we define a generic MIS model in Section 2, followed by a discussion of MDS2, R-GMA, and Hawkeye where we map each MIS to our generic model. Section 3 details the analysis and comparison results along with our suggestions for performance improvements. Section 4 reviews related work, and we conclude in Section 5.

2. Monitoring and Information System Overview

This section introduces our generic MIS model, describes NetLogger, and then discusses three monitoring and information systems: MDS2, R-GMA, and Hawkeye. For each system, we describe its background, the way the components map to our generic model, and the NetLogger instrumentation.

2.1. Generic MIS Model

Different MIS implementations employ diverse design patterns, underlying architectures, and implementation technologies. We define a generic MIS model to better compare the behavior of different MIS which has four service components, depicted in Figure 1. The *Information Collector* is a sensor, probe, or simple program that runs on a resource to generate data describing some property of that resource, for example, the CPU load, available disk space, or number of connections to a service. Typically, a single

Table 1: MIS component mapping to generic model.

	MDS2	R-GMA	Hawkeye
Information Collector	Information Provider	Producer	Module
Information Server	GRIS	ProducerServlet	Agent
Aggregate Information Server	GIIS	Compound Producer-Consumer	Manager
Directory Server	GIIS	Registry	Manager

resource runs many Information Collectors to publish a variety of static and dynamic information. The data is then collected and integrated by an *Information Server*, resulting in a snapshot of the state of a single resource and an easy way to collect a set of data. Data from a set of resources can be collected by using an *Aggregate Information Server*, which accepts registrations from multiple Information Servers, usually from all the resources at a site or set of sites. The Aggregate Information Server performs a higher-level management function between sites, whereas the Information Server is a resource-level service. In order to provide better scalability, Aggregate Information Servers are often organized into a hierarchical structure. Finally, the *Directory Server* component of our generic model is a resource registration service that provides resource lookup and discovery for clients. Table 1 shows a mapping from the generic model to each specific MIS discussed below.

The other common generic model for monitoring systems is the Grid Monitoring Architecture (GMA) [52], defined by the Global Grid Forum (GGF) [19]. GMA consists of three components, shown in Figure 2: Consumers, Producers, and a Registry. *Producers* are sources of data, and correspond to Information Collectors in our generic model. Producers register themselves with the *Registry*, which stores the details of

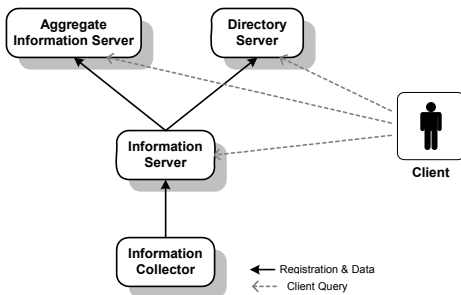


Figure 1: Generic MIS model consisting of four components.

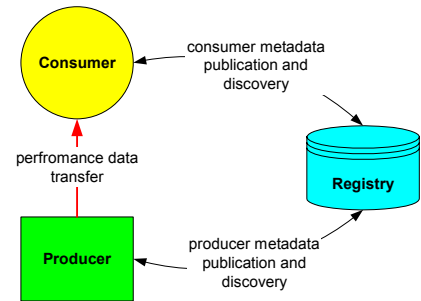


Figure 2: GMA components.

the producers, similar to the Directory Server in our generic model. This allows *Consumers*, software agents or users, to find the type of data created by a Producer and information about how to contact a specific Producer for that data. GMA as defined currently does not specify the protocols or the underlying data model to be used. All of the monitoring and information systems we examined included explicit aggregation at two levels, per resource and across resources, so we felt it necessary to have this explicitly defined in the model that we used to compare systems.

2.2. MIS Phases and Events

We also find that although each MIS is built very differently, the data queries made from a client to any component consist of three phases: the *connection phase* (client prepares the query and then builds a connection to the server-side service), the *processing phase* (server processes and answers the query), and the *response transmission phase* (server sends the query result back to the client).

In order to understand the time spent for each of these phases, we use NetLogger [21, 53] to gather data about the finer-grain “events” (basic units of monitoring data consisting of a name and timestamp) which each MIS uses to implement the query phases. The individual events do not map across the MIS systems, only the coarser-grained phases.

NetLogger is a toolkit developed at Lawrence Berkeley National Laboratory to monitor, under actual operating conditions, the behavior of elements of a complex distributed system or subsystem in order to determine where time is spent within such a system and identify the performance bottlenecks. With NetLogger, the software components running on NTP-synchronized hosts in a distributed system can be modified to produce time-stamped logs of interesting events are then correlated to allow a detailed performance characterization of the system components.

We used NetLogger to instrument all three MIS systems in our experimental study by inserting NetLogger calls into both the client-side and server-side source code of each system. We inserted a begin call and a corresponding end call for each event. However, we do not show all of the end events in our

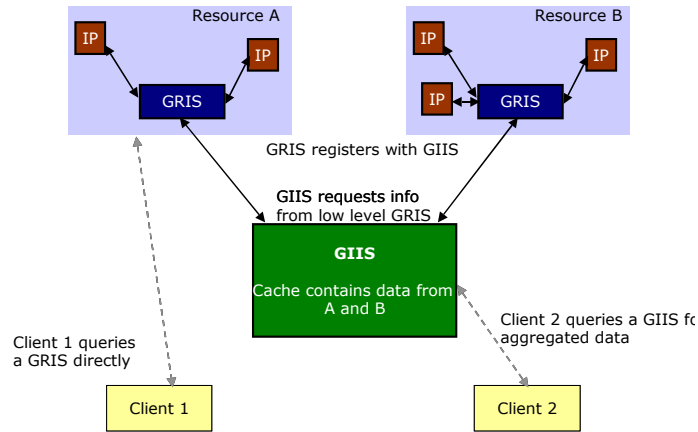


Figure 3: MDS2 architecture. There can be several levels of GIIS, and any GRIS or GIIS can register with another.

results because of the negligible time interval between the end call of one event and the begin call of the following event.

2.3. MDS2

The Monitoring and Discovery Service (MDS2) [10, 37] is the information service component of the pre-Web services Globus Toolkit® version 2. It is designed to provide an extensible framework for managing static and dynamic information about the status of a computational Grid and all its resources, and it is used primarily to address the resource selection problem. The MDS2 uses the Lightweight Directory Access Protocol (LDAP) [41, 57] as a uniform means of storing system information from a rich variety of system components, for constructing a uniform namespace for resource information across a system that may consist of many organizations, and for query processing. MDS2 also supports secure data access through the use of Grid Security Infrastructure (GSI) credentials [15].

MDS2 has a hierarchical structure that consists of three main components, shown in Figure 3. *Information Providers (IPs)* collect data about a resource or resource properties and correspond to the Information Collectors defined in our generic model. This data is aggregated at a per resource level by the *Grid Resource Information Service (GRIS)*, which acts as a content gateway for the resource, corresponding to the Information Server component. Across resources, the *Grid Index Information Service (GIIS)*

Table 2: MDS2 GRIS query instrumentation.

Step	Step Name	Step Process	Location	Generic Query Phase
1	<i>Client-Connect</i>	Client opens a connection to the GRIS.	Client	Conn.
2	<i>Client-SendQueryGRIS</i>	Client sends the query to the GRIS.	Client & Server	Conn.
3	<i>GRIS-InitSearch</i>	GRIS validates the query and initializes local data structures.	Server	Proc.
4	<i>GRIS-SearchIndex</i>	GRIS searches index to determine what IP(s) to query.	Server	Proc.
5	<i>GRIS -InvokeIP</i>	GRIS sends request to IP(s) for fresh data and receives results.	Server	Proc.
6	<i>GRIS-GenResult</i>	GRIS integrates results into a single reply.	Server	Proc.
7	<i>Client-ReceiveResult</i>	Client receives results and disconnects.	Server & Client	Trans.

aggregates information from the lower-level GRISes, thus playing the role of the Aggregate Information Server in the generic model. The GIIS also provides a directory lookup service for Grid users and applications, corresponding to the Directory Server in the generic model. In MDS2, each service component registers with others using a soft-state protocol that allows dynamic cleaning of dead resources. Both the GRIS and GIIS cache their data to minimize I/O and network overhead. Tables 2 and 3 list the events we log for a GRIS and GIIS query, respectively.

Table 3: MDS2 GIIS query instrumentation.

Step	Step Name	Step Definition	Location	Generic Query Phase
1	<i>Client-Connect</i>	Client opens a connection to the GIIS.	Client	Conn.
2	<i>Client-SendQueryGIIS</i>	Client sends the query to the GIIS.	Client & Server	Conn.
3	<i>GIIS-InitSearch</i>	GIIS validates the query and initializes local data structures.	Server	Proc.
4	<i>GIIS-SearchIndex</i>	GIIS searches index to determine what GRIS(s) to query.	Server	Proc.
5	<i>GIIS-InvokeGRIS</i>	GIIS requests fresh data from the GRIS(s).	Server	Proc.
6	<i>GIIS-GenResult</i>	GIIS integrates results into a single reply.	Server	Proc.
7	<i>Client-ReceiveResult</i>	Client receives result and disconnects.	Server & Client	Trans.

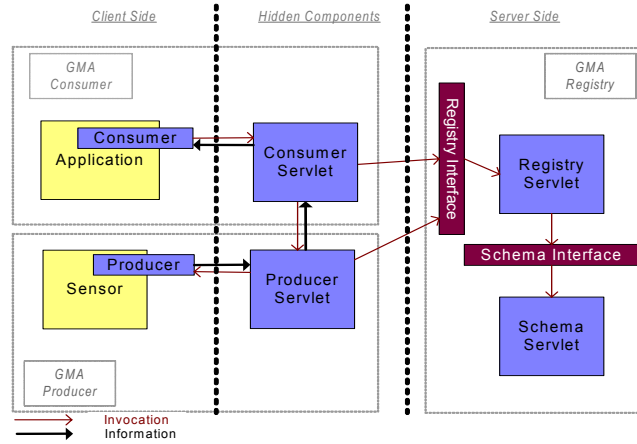


Figure 4: R-GMA architecture mapped to GMA components.

2.4. GMA and R-GMA

The Relational Grid Monitoring Architecture (R-GMA) [8, 11], was developed to provide monitoring services for the European DataGrid project. R-GMA, implemented with Java Servlet Technology [27], builds on the GMA, described earlier, but in addition specifies a relational data model [14] and an SQL-based query language. The main use of R-GMA is for notification of events—that is, a user subscribes to a flow of data directly from a data source and receives a continuous datastream from that source.

Figure 4 illustrates the R-GMA components and their mapping to GMA. When a *Producer* is created, its registration details are sent via a *ProducerServlet* to the *Registry*, similar to an Information Collector registering with the Directory Server in our generic model. The Registry records details about the Producer, including the description and view of the data published, but not the data itself. When the Producer publishes data, the data is transferred to a local *ProducerServlet*. When a *Consumer* is created, its registration details are also sent to the Registry via a *ConsumerServlet*. The Registry records the type of data that the Consumer is interested in and then returns a list of Producers back to the *ConsumerServlet* that matches the Consumer’s selection criteria. The *ConsumerServlet* then contacts the relevant *ProducerServlet*s to initiate a transfer of data from them. The data is then available to the Consumer through the *ConsumerServlet*.

Table 4: R-GMA Producer query instrumentation.

Step	Step Name	Step Definition	Location	Generic Query Phase
1	<i>Client-StartConsumer</i>	Client creates a Consumer instance.	Client	Conn.
2	<i>Consumer-Start</i>	Consumer prepares to query the local ConsumerServlet.	Client	Conn.
3	<i>ConsumerServlet-Start</i>	ConsumerServlet initializes local data structures.	Client	Conn.
4	<i>ConsumerServlet-StartQuery</i>	ConsumerServlet sends query to ProducerServlet.	Client & Server	Conn.
5	<i>Producer-Execute</i>	ProducerServlet queries the Producer, and the result is returned to the ProducerServlet.	Server	Proc
6	<i>Client-Check</i>	Client checks query status until a complete message is received, indicating data has been transferred from the ProducerServlet to the local store of the ConsumerServlet.	Server & Client	Trans.
7	<i>Client-Retrieve</i>	Client requests data from the Consumer.	Client	Trans.
8	<i>Consumer-Retrieve</i>	Consumer requests data from Consumer Servlet.	Client	Trans.
9	<i>ConsumerServlet-Retrieve</i>	ConsumerServlet retrieves the data from its local store.	Client	Trans.
10	<i>Consumer-DoneRetrieval</i>	Consumer receives the retrieved data from the ConsumerServlet.	Client	Trans.
11	<i>Client-DoneRetrieval</i>	Client receives the retrieved data from Consumer.	Client	Trans.

Table 5: R-GMA Registry query instrumentation.

Step	Step Name	Step Definition	Location	Generic Query Phase
1	<i>Client-PrepareQuery</i>	Client prepares a query to register a new Consumer with the Registry and contacts the ConsumerServlet.	Client	Conn.
2	<i>ConsumerServlet-SendQueryRegistry</i>	ConsumerServlet connects and sends the query to the RegistryServlet.	Client & Server	Conn.
3	<i>RegistryServlet-ProcessQuery</i>	RegistryServlet registers the new Consumer and finds related Producers.	Server	Proc.
4	<i>ConsumerServlet-ReceiveResults</i>	ConsumerServlet receives query results from the RegistryServlet.	Server & Client	Trans.
5	<i>Client-ReceiveResults</i>	Client retrieves query results from the local ConsumerServlet.	Client	Trans.

R-GMA has two main types of producers: *StreamProducers*, which output a continuous stream of data by pushing data to the Consumer and *LatestProducers*, which publish only the latest value held when queried, using a pull approach. R-GMA also allows the creation of *compound Producer-Consumers* to define higher-level services. For example, the new R-GMA archiver consists of a Consumer module that subscribes

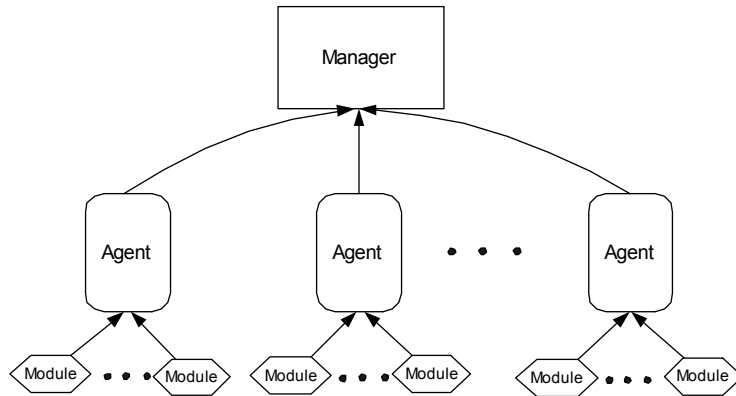


Figure 5: Summary of the Hawkeye architecture.

to a set of datastreams produced by StreamProducers, a backend database store, and a Producer front-end that advertises the newly combined archive data for consumption by other consumers. Tables 4 and 5 detail a producer and a registry query, respectively.

2.5. Hawkeye

Hawkeye [23] is a monitoring and management system that uses Condor [35] and ClassAd technologies [46] for its underlying infrastructure. Hawkeye was designed to automate problem detection by publishing monitoring data that is then used to trigger executables in response to matching predefined conditions, for example to identify high CPU load, increasing network contention, or resource failure within a distributed system. Hawkeye involves two fundamental ideas: its use of the Condor *ClassAds* [46], sets of attribute/value pairs (e.g., “operating system” and “Linux”), to communicate resource information, and *ClassAd Matchmaking* [47] to match defined problem cases to existing resource conditions.

Table 6: Hawkeye Agent query instrumentation.

Step	Step Name	Step Definition	Location	Generic Query Phase
1	<i>Client-PrepareQuery</i>	Client constructs a query and locates the Manager.	Client	Conn.
2	<i>Client-LocateAgent</i>	Client retrieves the Agent address from Manager.	Client	Conn.
3	<i>Client-ConnectAgent</i>	Client establishes a connection with the Agent.	Client	Conn.
4	<i>Client-SendQueryAgent</i>	Client sends query to the Agent.	Client & Server	Conn.
5	<i>Agent-AnswerQuery</i>	Agent finds ClassAds that satisfy the query.	Server	Proc.
6	<i>Agent-SendClientAnswer</i>	Agent sends the query results to the client.	Server & Client	Trans..
7	<i>Client-ProcessResults</i>	Client disconnects from the server and formats the results.	Client	Trans.

The architecture of Hawkeye is shown in Figure 5. A *Module* is a sensor that advertises resource information, and corresponds to Information Collectors in our generic model. An *Agent* receives ClassAds from a set of Modules and integrates them into a single ClassAd, equivalent to the Information Server component of the generic model. This integration uses “passive caching” to keep the data up to date. Each Module updates its data to the Agent at one-minute intervals in the form of full ClassAds, which are combined into a single ClassAd. A *Manager* is the central collection point for a set of resources, similar to the Aggregate Information Server in the generic model, and it collects and stores (in an indexed resident database) monitoring information from each Agent registered to it. The Manager is also the central target for

Table 7: Hawkeye Manager query instrumentation.

Step	Step Name	Phase Definition	Location	Generic Query Phase
1	<i>Client-PrepareQuery</i>	Client constructs a query and locates the Manager.	Client	Conn.
2	<i>Client-ConnectMgr</i>	Client establishes a connection with the Manager.	Client	Conn.
3	<i>Client-SendQueryMgr</i>	Client sends query to the Manager.	Client & Server	Conn.
4	<i>Mgr-ProcessQuery</i>	Manager searches ClassAds that satisfies the query.	Server	Proc.
5	<i>Client-ProcessResults</i>	Client receives the query results from the Manager, disconnects from the Manager, and formats the results.	Server & Client	Trans.

queries about the status of any resource and, at fixed intervals, is sent data from each registered Agent, similar to the generic model Directory Server. Tables 6 and 7 detail an Agent query and a Manager query, respectively.

3. Experiment Results and Evaluation

In this section we describe the experimental setup and the performance metrics we gathered from the experimental systems. We then analyze and compare the results of the experiments. Raw data results can be found online at <http://www.cs.wisc.edu/~jfreschl/MISExperimentData/>.

Our benchmarking experiments address the following questions:

- *How does the performance of an Information Server scale with the number of users?* As one of the most heavily-accessed service components in an MIS, the information server should scale well with a large number of concurrent users.
- *How does the performance of a Directory Server scale with the number of users?* The importance of this study comes from the anticipation that the number of the users depending on directory servers to discover and locate resources will increase dramatically as deployed Grid testbeds grow in size.
- *How does the performance of an Information Server scale with the number of Information Collectors?* New Information Collectors are constantly being identified and developed for MIS deployments, and information servers will need to be able to scale to handle the additional load and management. Some monitoring systems (such as WatchTower [29] or Inca [24, 50]) can publish more than 300 individual pieces of information.
- *How does an Aggregate Information Server scale with the number of Information Servers it is aggregating?* Understanding this aspect of the MIS will help us understand how to build hierarchies of aggregation and still achieve good performance, which will be a factor as it becomes more common to aggregate data of interest from a larger number of sites.

3.1. General Experiment Setup

We ran our experiments between two sites: the Lucky testbed at Argonne National Laboratory (ANL), which provided the server-side services of MDS2, R-GMA, and Hawkeye, and a testbed at the University of Chicago (UC), which provided the client-side services. We used a LAN setting for both sides so that the network factors would be relatively similar and uniform, and network times would not be the dominating factor in the experiments; rather we could focus on the software factors.

The Lucky testbed comprised seven Linux machines with hostnames *lucky{0,1,3,..., 7}.mcs.anl* (*lucky2* was unavailable during the experiments) and a shared file system on a 100 Mbps LAN. Each machine was equipped with two 1133 MHz Intel PIII CPUs (with a 512 KB cache per CPU) and 512 MB RAM. *Lucky0* and *lucky6* ran Linux kernel 2.4.10, and the rest ran kernel 2.4.19.

The UC client-side hosts comprised a cluster of 20 Linux machines with a shared file system on a 100 Mbps LAN. Fifteen of them were equipped with a 1208 MHz CPU and 256 MB RAM, while the rest had a slightly slower CPU (but at least 756 MHz), also with 256 MB RAM. Each machine ran a Linux 2.4 kernel with version number 2.4.17 or higher.

The bandwidth between ANL and UC was around 55 Mbits per sec on average (as measured by Iperf [25]), and the latency (round-trip time) was 2.3 msec on average.

We deployed MDS2 version 2.4, R-GMA version 3.4.6 (with a MySQL [39] version 3.23.49 backend database), and Hawkeye version 1.0 RC5 on both sites, without security. We deployed the packaged Information Collectors for MDS2 and Hawkeye, and created a customized LatestProducer to generate the host's CPU load information every one minute for R-GMA which doesn't ship with standard IPs. We used NetLogger version 2.2.6 to instrument all the releases of the three services. To synchronize the clock, we ran NTP [40] version 4.1.2 on the Lucky testbed and NTP 4.0.91 on the UC client hosts.

In our experiments, we simulated up to 600 users by running individual user processes (scripts) distributed evenly across the 20 UC client-side hosts. For R-GMA we configured the client-side machines so that each client machine ran its own ConsumerServlet to handle the requests from all the Consumer instances

running on that machine instead of using a global ConsumerServlet shared with other machines in order to reduce the communication cost between the Consumers and a non-local ConsumerServlet.

The values reported in each experiment are the average over a 10-minute time span. We used Ganglia [17], a cluster monitoring system developed at UC Berkeley, to collect the performance data at five-second intervals. All user requests to the server-side service components in MDS2, R-GMA, and Hawkeye are run with a one-second *wait period*. That is, after a user queries a service component and receives a response, the user waits one second before sending its next query. Note this does not mean that queries are sent once a second; rather, this is equivalent to blocking sends with a one-second wait between each.

3.2. Performance Metrics

To evaluate the performance of MDS2, R-GMA and Hawkeye, we used five performance metrics: throughput, client-side observed response time (CORT), server-side request processing time (SRPT), load1, and CPU-load.

Throughput is defined as the average number of requests, or queries, processed by an MIS component per second. The higher the throughput value achieved, the better the performance.

Client-side observed response time (CORT), which is equivalent to the metric response time used in our previous work [60], denotes the average amount of time, in seconds, from the point a user sends out a request till the user gets the response. The CORT is calculated at the client-side.

Server-side request processing time (SRPT), is the sum of all the sever-side NetLogger phases and represents the time for the server-side components to process the user query. As a superset, CORT is always greater than SRPT. The smaller the CORT or SRPT value, the better the performance.

We also used two load metrics for the experiments; load1 and CPU load. *Load1*, also called the one-minute load average, is the average number of processes in the ready queue waiting to run over the last minute measured by the Ganglia metric *load_one*. *CPU load* indicates the percentage of the CPU cycles spent in user mode and system mode, which we measured by adding the *cpu_user* and *cpu_system* values recorded by Ganglia. CPU load may be high while load1 is low if a machine is running a small number of

compute-intensive applications. CPU load may be low while load1 is high if the same machine is trying to run a large number of applications that are blocking on I/O.

3.3. Experiment Set 1 – Information Server Scalability with Users

In the first experiment set, we evaluated the scalability of an Information Server with respect to the number of concurrent users. Specifically we examined the MDS2 GRIS, the R-GMA ProducerServlet, and the Hawkeye Agent. Since the Information Server is the most heavily queried component in an MIS, it is important to understand its efficiency.

3.3.1 Experimental Setup

For MDS 2.4, we ran a GRIS on *lucky7* with ten reporting Information Providers (IPs). We examined two different scenarios: the GRIS always caching the data from the IPs and the GRIS never caching the data (and therefore having to run the IPs for each query). By understanding the GRIS performance under two extreme conditions we can estimate the performance of the average case, which is somewhere between them. Each query requested all the data elements in the GRIS directory generated by all 10 IPs. The average size of requested data was less than 10 KB.

For R-GMA, we ran a ProducerServlet (with type LatestProducerServlet) at *lucky3* managing 10 local Producers, and a Registry on *lucky1*. Each Consumer queried for all CPU load data generated by all 10 Producers. The combined data size for the request for all the data was less than 2 KB. To force the Consumers (via their local ConsumerServlet) to directly request the ProducerServlet for data instead of consulting the Registry first to find out the proper Producer connections, we passed the ten Producer connections explicitly to each Consumer, thereby avoiding the Registry's involvement in each query.

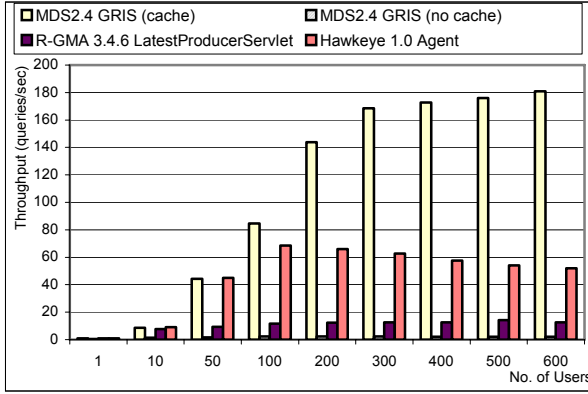


Figure 6: Information Server throughput compared to the number of users.

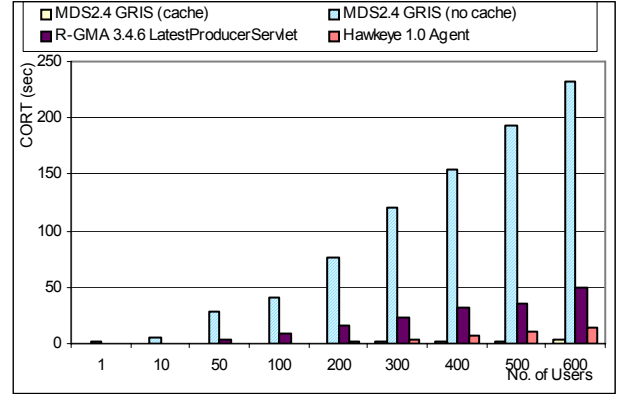


Figure 7: Information Server CORT compared to the number of users.

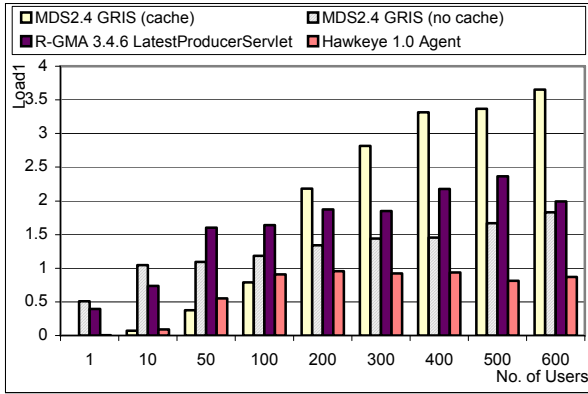


Figure 8: Information Server host load1 compared to the number of users.

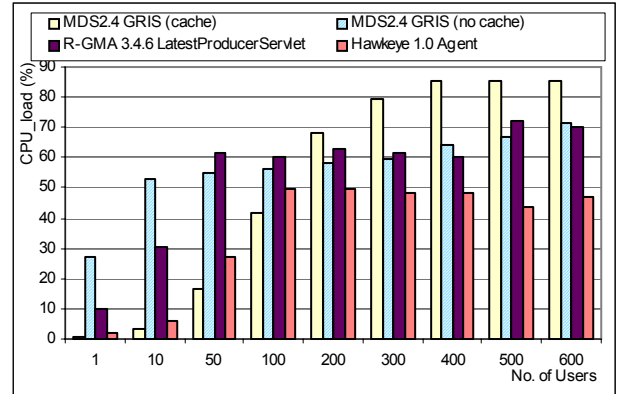


Figure 9: Information Server host CPU-load compared to the number of users.

For Hawkeye, we ran an Agent on *lucky5* with 10 Modules registered to it and a Hawkeye Manager on *lucky3*. Each query requested for all of the information in the ClassAd published by the *lucky5* Agent. The average size of the requested data was about 10 KB.

Figures 6–11 show the performance results of the three Information Servers including two different data caching scenarios for MDS 2.4 GRIS. Figure 10 shows the NetLogger instrumentation results when queried by 50 and 400 users or Consumers, and Figure 11 shows a summary of the generic query phases.

3.3.2 Experimental Results

As expected, MDS2 with caching scaled much better than its non-caching counterpart, since the later executes the high cost low-level IPs that also compete with each other. For a GRIS with data caching, most

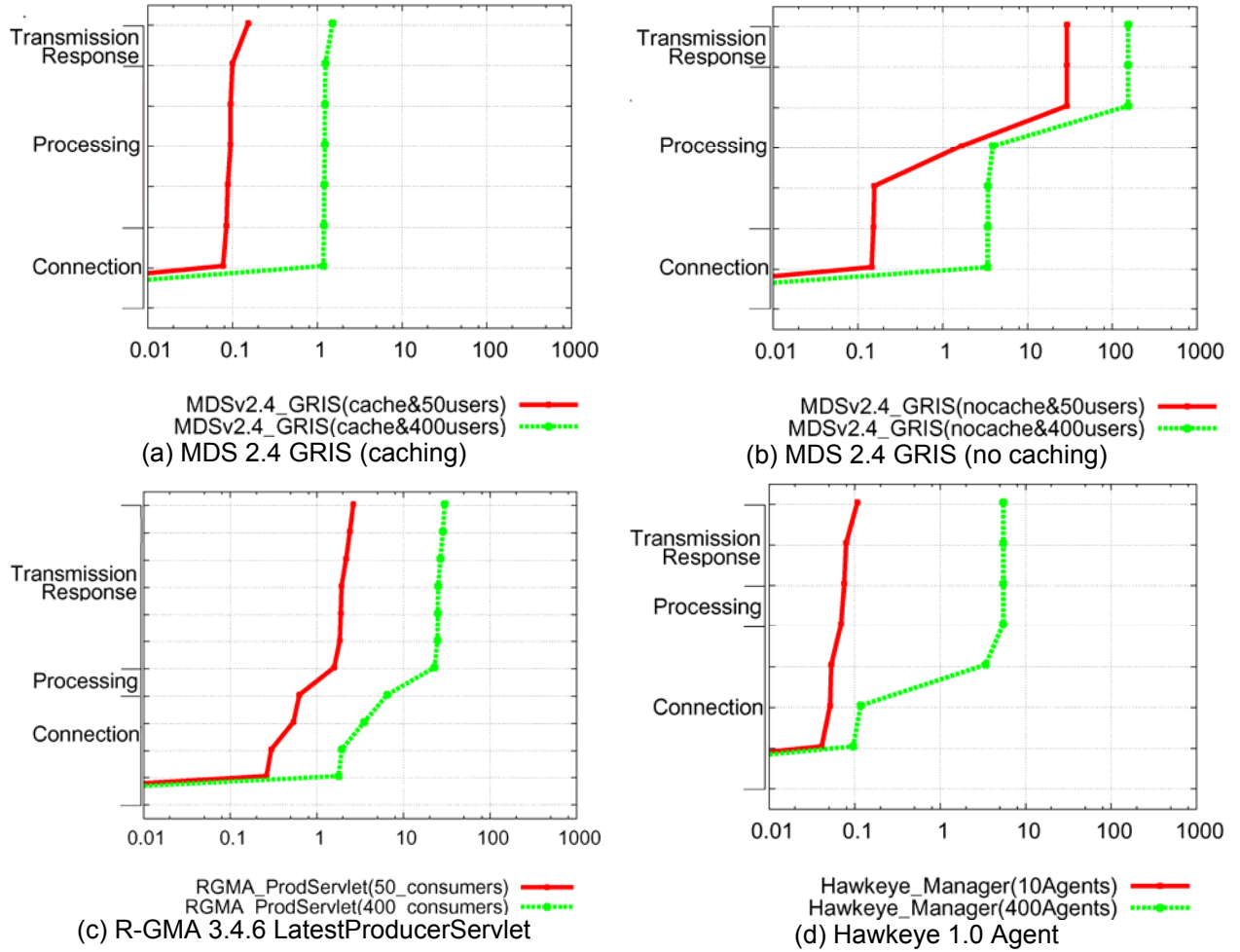


Figure 10: Information Server performance by NetLogger step for 50 and 400 users.

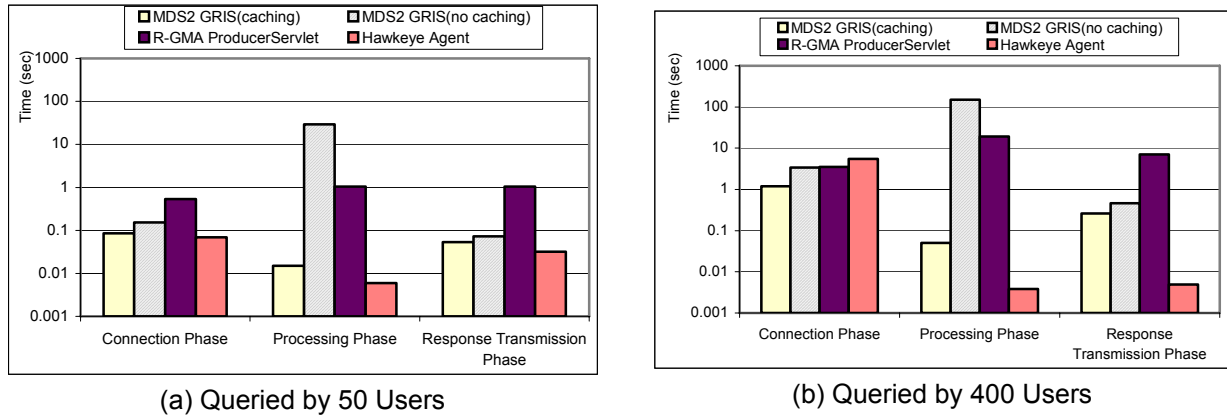


Figure 11: Information Server generic query phase comparison.

time of the CORT is spent on building a connection to the GRIS on the client-side possibly due to network limitations on the server side and the constraints of the LDAP protocol used by MDS2.

R-GMA performance decreases significantly when there are more than 100 Consumers (throughput of 12 queries per second). The processing and transmission phases for R-GMA have poor performance when compared with the other systems. The main cause is the query preparation and sending. To retrieve data from each Producer, the client-side ConsumerServlet sends a new connection request to the ProducerServlet, which then builds a new database connection to query the data from that Producer. Creating a new database connection to query each Producer is an expensive operation, especially when a large number of connections exist. Though MySQL databases for Linux can support thousands of concurrent connections in theory, in practice it cannot handle nearly as many concurrent connections in that each connection uses memory and CPU [59]. When the MySQL database resides on a non-dedicated resource, the performance can degrade further.

In general, the Hawkeye Agent scaled well with increasing numbers of users, and the Hawkeye processing phase is the most efficient of the three systems. The main cost was communication - when a client connects to the Agent, a new connection must be established because Hawkeye does not allow socket caching from previous queries. Hawkeye's relatively stable load is likely due to the fact that the Agent is single threaded, so no matter how many concurrent users query the Agent, only one can successfully connect to the Agent at a time.

3.4. Experiment Set 2 – Directory Server Scalability

The second functionality we tested was the scalability of a Directory Server with respect to the number of concurrent users. In particular, we examined the performance of the MDS2 GIIS, the R-GMA Registry, and the Hawkeye Manager. The number of users consulting a Directory Server to discover and locate distributed resources will increase dramatically as deployed Grid testbeds grow in size.

3.4.1 Experimental Setup

We configured the Lucky testbed to run an MDS 2.4 GIIS on *lucky0* and a GRIS with 10 IPs on the other Lucky nodes, for a total of 60 IPs over the testbed. To analyze only the directory functionality of the GIIS, and not its information aggregation capacity, we set the cache element time to live (cachettl) parameter to a very large value so that the data was always in the cache. The user queries were the same as those used in Experiment 1.

For R-GMA, we ran a Registry at *lucky1* and one ProducerServlet with 10 local LatestProducers on each of the other Lucky nodes so that the Registry stores the registration information from 60 Producers. Since we wanted to examine the scalability of the Registry, we did not pass any Producer connections to the Consumers as we did in the first set of experiments, thereby forcing a Consumer to consult the Registry for the locations of Producers, as shown in Table 5. The user queries were the same as those used in Experiment 1.

For Hawkeye, we ran the Manager at *lucky3* and up to 600 users concurrently querying the Manager. There were 6 Agents (one on each Lucky node) each running 10 default Modules, 60 Modules overall. The query we used was for all information contained in the 6 ClassAds published by the Lucky nodes, giving an average result size of 60 KB.

Figures 12–17 show the performance results of the three Directory Servers. The end-to-end performance results are presented in Figures 12–15; Figure 16 shows the NetLogger instrumentation results for each Directory Server when accessed by 50 and 400 users, using a log scale for the x-axis. Figure 17 shows a summary comparison with respect to the generic query phases.

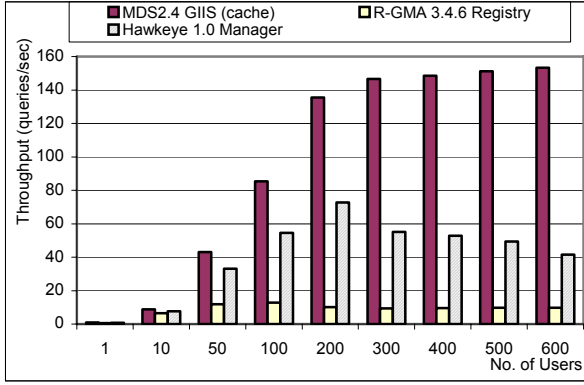


Figure 12: Directory Server throughput compared to the number of users.

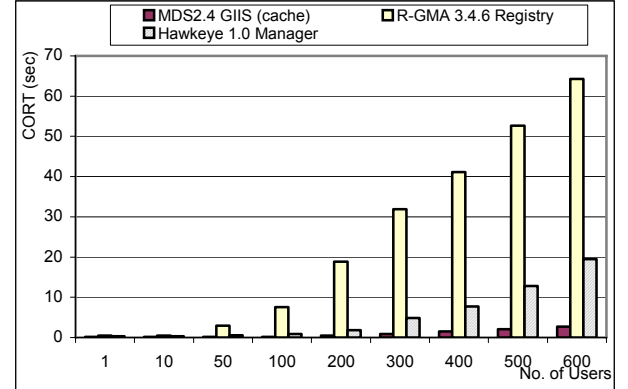


Figure 13: Directory Server CORT compared to the number of users.

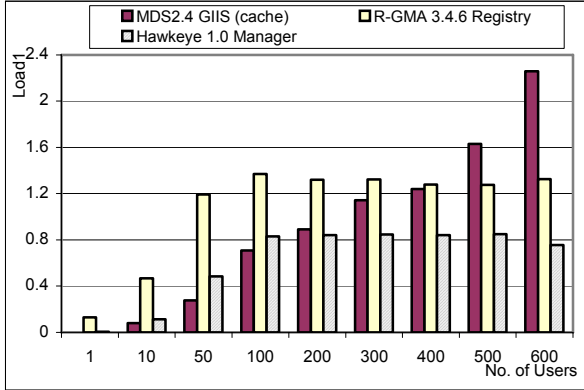


Figure 14: Directory Server host load compared to the number of users.

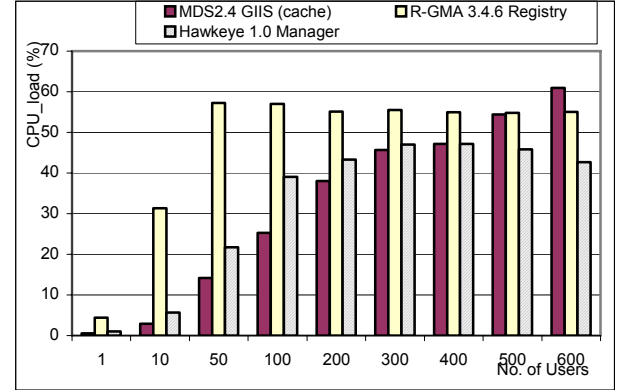


Figure 15: Directory Server host CPU-load compared to the number of users.

3.4.2 Experimental Results

The MDS GIIS performance generally scales well with respect to the number of users. The majority of a query's CORT is spent in the client connection step, step 1 in Table 3. In general, a GRIS is more efficient in serving queries than a GIIS because it has fewer entries, therefore shorter searching times.

The R-GMA Registry saw significant performance and scalability problems in this set of experiments. The RegistryServlet invoking the Registry instance, step 3 in Table 5, makes up almost 90% of the CORT. The performance of all client-side phases vary little in both test cases, indicating that the scalability constraint lies on the server side. The Registry relies on a continuous database connection to handle requests

from all Consumers rather than creating a new connection for each request. To serve a Consumer query for all qualified Producers, however, the Registry first removes any expired Producers and Consumers from the database, and then it issues an SQL query to search for relevant Producers, which requires a lock on the related database tables, and the associated delay.

The end-to-end performance results for the Hawkeye Manager are similar to the results seen for the Agent in Experiment 1. The load is stable, and the most noticeable change as the number of users increases is in the connection times. The only significant difference in the Agent and Manager code is that the Manager can be multithreaded, which could decrease user contention, and explain the slight difference between these results and those in Experiment 1.

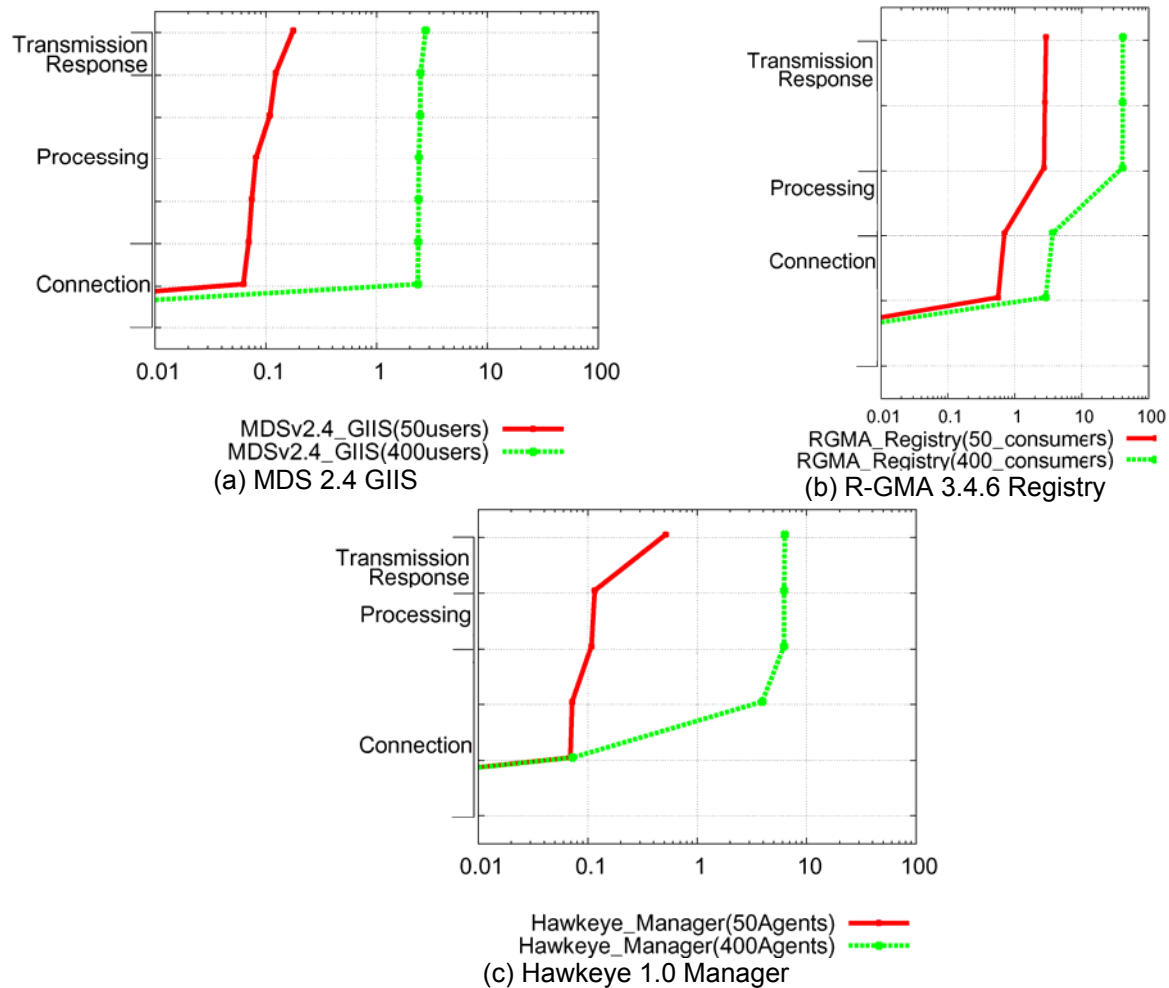
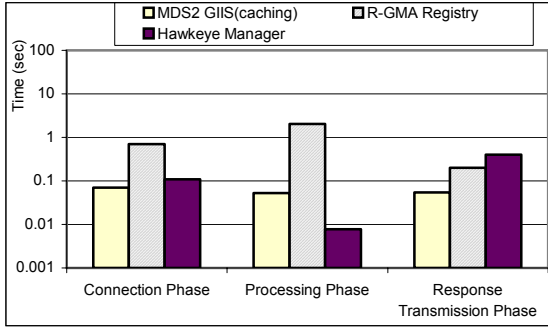
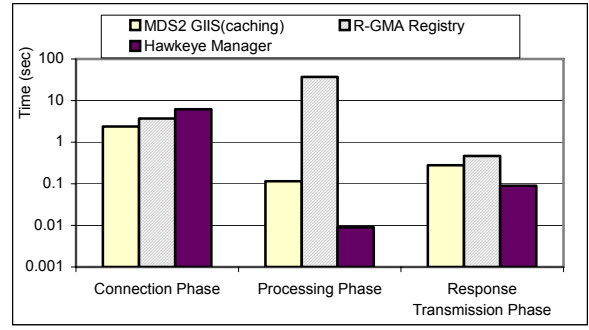


Figure 16: Directory Server performance by NetLogger step for 50 and 400 users.



(a) Queried by 50 Users



(b) Queried by 400 Users

Figure 17: Directory Server generic query phase comparison.

3.5. Experiment Set 3 – Information Server Scalability with Information Collectors

Our third set of experiments evaluated how the performance of an Information Server scaled by the number of the Information Collectors registered to it. Specifically, we examined the MDS2 GRIS and its Information Providers (IPs), the Hawkeye Agent and its Modules, and the R-GMA ProducerServlet and Producers. The reason we investigate this aspect of the MIS performance is that the information server must have the potential to support increasing demands for more Information Collectors.

3.5.1 Experimental Setup

For the MDS2 experiments, we modified the default memory IP and added copies of the new version to simulate the extra IPs. An MDS 2.4 GRIS was run at *lucky7*, and 10 client-side users sent queries to it for the information from all the IPs. We test 10 to 90 IPs reporting to the same GRIS. To evaluate the effect of the data caching of the GRIS, we also simulated two extreme caching scenarios: the data always in the GRIS cache and never in the GRIS cache. Since we queried for the memory information from every element in the directory, the average size of request data was between 10 and 100 KB.

For R-GMA, we ran a ProducerServlet of type LatestProducerServlet at *lucky3* and a Registry on *lucky1*. We varied the number of the local Producers of type LatestProducer from 10 to 90 and used 10 Consumers to send concurrent queries. Each Consumer requested all the data generated by all the Producers, and the

combined data size varied with the number of Producers, from 20 KB to 180 KB. To ignore the effect of consulting the Registry to locate the qualified Producers, we passed all the Producer connections explicitly to each Consumer, similar to Experiment 1.

For Hawkeye, we varied the total number of Modules running on each pool member from the 10 default Modules to 90 using multiple instances of the “memory” Module. Once the Modules were running on each pool member, 10 users queried the Manager. The query we used was for all information contained in the ClassAds published by all Modules, since Hawkeye cannot return individual ClassAd fields. The Hawkeye system passive caching also played a role in this experiment. Instead of actively updating data, such as the MDS does, each Module updated its data to the Agent independently at one-minute intervals, whether or not it was out of date, so an Agent did not generate fresh data, and no data in its database was more than a minute out of date. The maximum number of Modules able to register to an Agent was 98: adding another Module caused the Hawkeye to crash because of current code limitations.

Figures 18–23 show the performance results of the three information servers scaling with the Information Collectors. Figure 22 describes the NetLogger instrumentation results when each information server manages 30 and 80 Information Collectors and uses a log scale for the x-axis. Figure 23 shows a summary comparison with respect to the generic query phases.

3.5.2 Experimental Results

The performance of the MDS2 GRIS without data caching degrades dramatically as the number of IPs increases, however, it can still achieve a throughput of 8 queries per second with a CORT of less than 0.1 second for 90 IPs if their data is cached. This result is compatible to the findings in our previous work [60]. When the GRIS doesn’t cache data, over 90% of the CORT is spent on the server side performing a request for data from the Information Providers, step 5 in Table 2, as opposed to the client connection bottleneck seen when data is cached.

The R-GMA ProducerServlet performance degrades dramatically when the number of Producers grows due to the high cost for the ProducerServlet to query each Producer, steps 4 and 5 in Table 4. The reason is

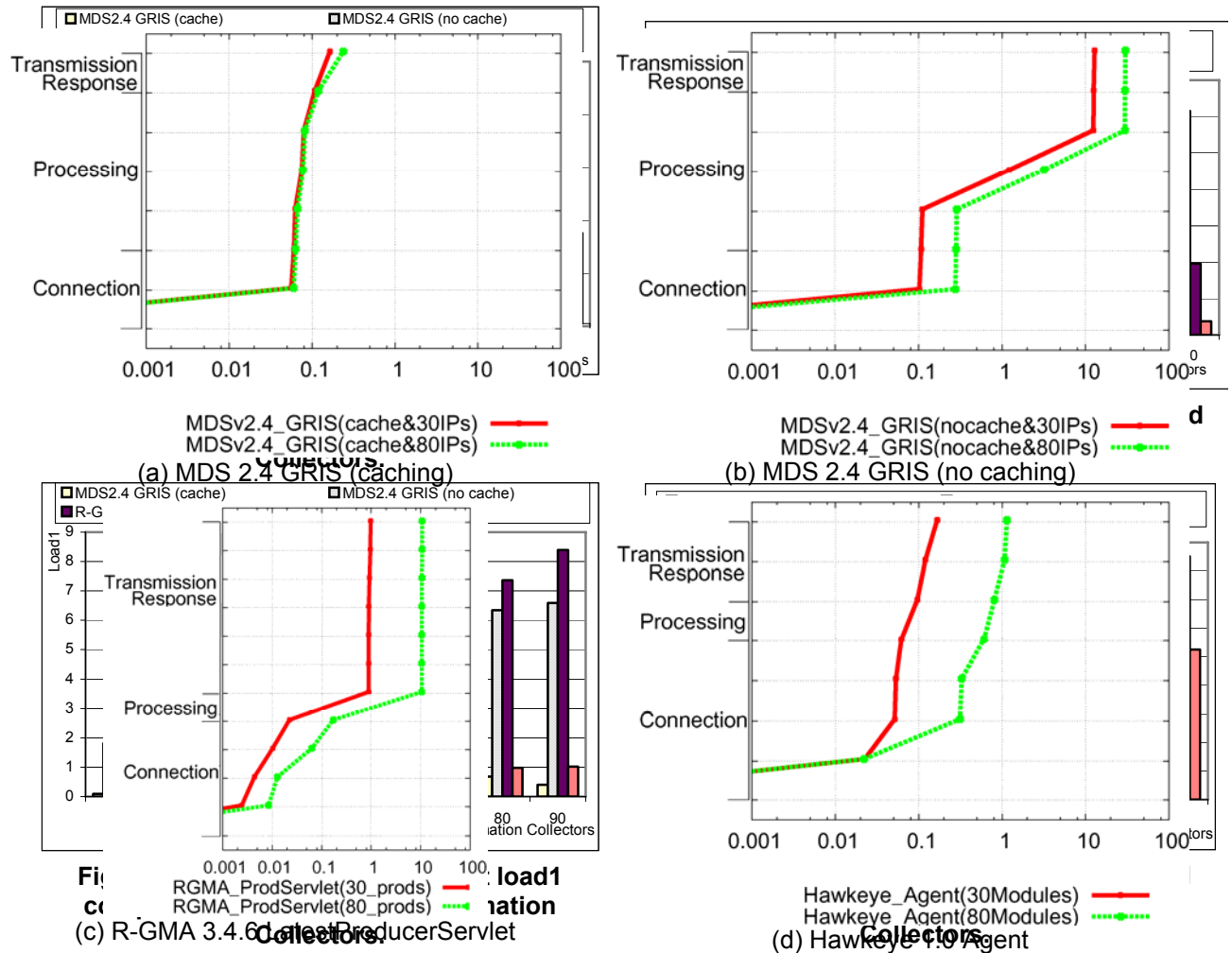


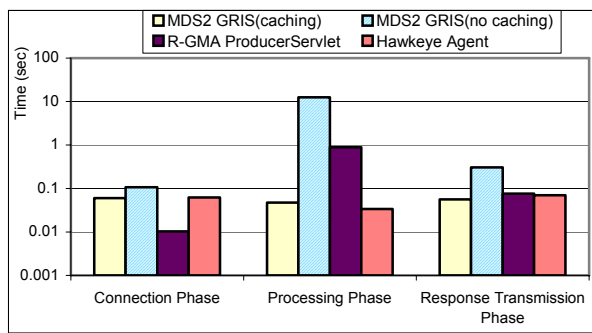
Figure 22: Information Server performance by NetLogger step for 30 and 80 Information Collectors

that although the number of the Consumers is fixed to 10 in all the experiments, the number of Producer connections passed to each Consumer is equal to the number of the registered Producers. For each Producer connection, the Consumer sends a separate connection request, via ConsumerServlet, to the ProducerServlet, and the ProducerServlet then builds a separate database connection to retrieve data from that Producer. More Producers managed by the ProducerServlet means additional database connections must be created to serve each query.

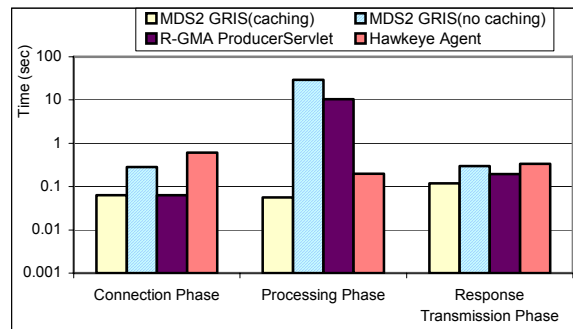
For Hawkeye, the Agent performance scales relatively well with a maximum CORT. In Figure 21 we see that adding modules does add extra CPU-load, but this does not seem to be a significant problem. When compared to the other systems, the Manager has similar performance in the communication phases and has the second best processing performance. The machine hosting the Hawkeye Agent meets a relatively constant load¹ regardless of the number of Modules being managed since the Agent does not run any module processes to generate fresh data, which also describes the similarity to the results seen in Figure 22(a) for MDS with caching.

3.6. Experiment Set 4 – Aggregate Information Server Scalability

In the last set of experiments, we evaluated the scalability of the Aggregate Information Server with respect to the number of registered Information Servers. Specifically we examined the MDS2 GIIS varying the number of GRIS, an R-GMA compound Producer-Consumer varying the number of Producers, and the Hawkeye Manager varying the number of Agents. Our objective was to better understand the upper limit of the number of Information Servers that should register to an Aggregate Information Server to be able to understand performance limitations as deployments grow.



(a) Registered by 30 Information Collectors



(b) Registered by 80 Information Collectors

Figure 23: Information Server generic query phase comparison.

3.6.1 Experimental Setup

For MDS 2.4, we simulated extra Information Servers by running multiple GRIS instances on each Lucky node, except on *lucky0* where the GIIS ran. We tested up to 600 GRIS instances, evenly distributed across all nodes. During the experiment, 10 users from the UC testbed concurrently sent queries to the GIIS for 10 minutes. We tested two kinds of queries: the first queried for all of the data available from each of the registered GRIS, and the second asked for only a portion of the data (memory information) from each registered GRIS. The requested data size varies from 10 KB to 6 MB when querying for all the data, and from 1 KB to 600 KB when querying for a portion of data.

For R-GMA, we ran a compound Producer-Consumer on *lucky3*, a Registry on *lucky1*, and up to 600 StreamProducers to interact with the compound Producer-Consumer on the rest of the Lucky nodes. Queries were performed by 10 Consumers from UC client-side for all the aggregated information at one-second intervals. The requested data size varied from 0.2 KB to 120 KB when aggregating all 600 StreamProducers. To evaluate the aggregation performance of the compound Producer-Consumer in isolation, we passed the connection of the Producer-Consumer directly to the 10 Consumers, preventing them from consulting the Registry to locate this service.

For Hawkeye, we simulated the large number of Agents by evenly distributing newly spawned Agents across the Lucky testbed on all nodes but *lucky3*, which ran the Manager. Once the simulated Agents were running, 10 concurrent users queried the Manager for information, which resulted in a query response of 10KB to 6MB depending on how many ClassAds were returned. We then had the Manager search the ClassAds using a query that matched all ClassAds.

Figures 24–29 show the performance results of the three Aggregate Information Servers scaling with the number of Information Servers. The end-to-end performance results are presented in Figures 24–27; Figure 28 compares the NetLogger instrumentation results when each Aggregate Information Server aggregates 50 and 400 registered information servers respectively using a log scale for the x-axis. Figure 29 shows a summary comparison with respect to the generic query phases.

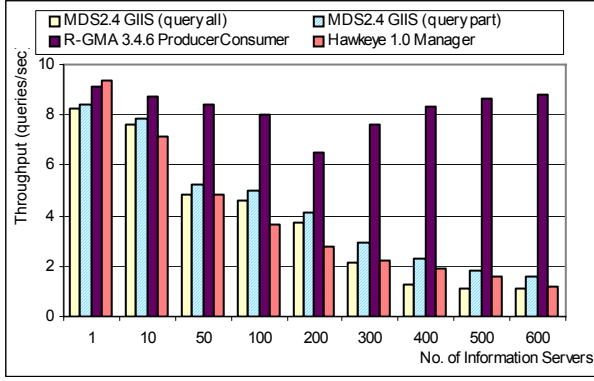


Figure 24: Aggregate Information Server throughput compared to the number of Information Servers.

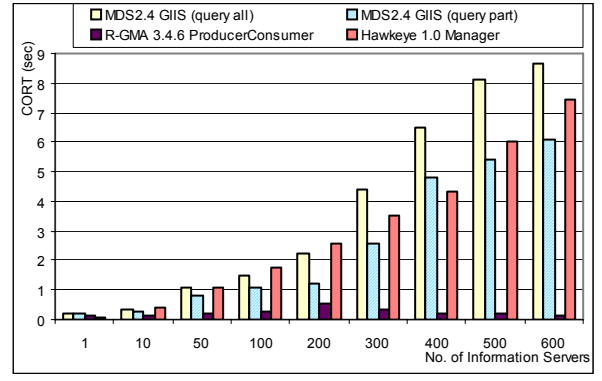


Figure 25: Aggregate Information Server CORT compared to the number of Information Servers.

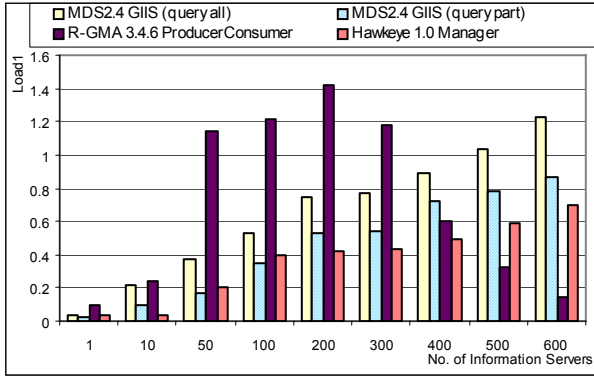


Figure 26: Aggregate Information Server host load1 compared to the number of Information Servers.

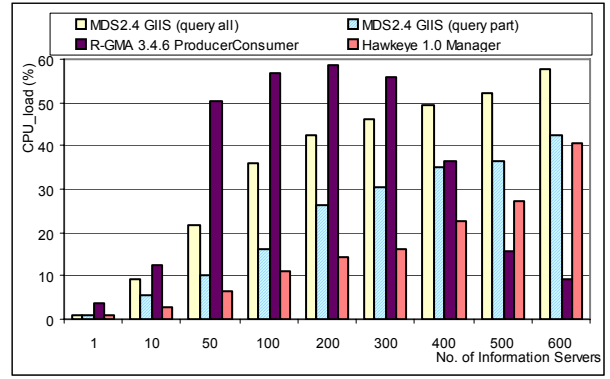


Figure 27: Aggregate Information Server host CPU-load compared to the number of Information Servers.

3.6.2 Experimental Results

The MDS2 GIIS encounters large performance degradation in both throughput and CORT due to the server queries to the GRIS which trigger additional queries to the IPs. We also observe a high load (both load1 and CPU-load) on the machine running the GIIS indicating that the contention grows as the number of GRISes grows.

We saw very little effect on the throughput and CORT for the R-GMA Producer-Consumer as we increased the number of aggregated Producers due in part to the use of a StreamProducer instead of a LatestProducer, as was deployed for the other experiments. In order to aggregate the data, StreamProducers

were needed because they are the only producers that can be consumed by the compound Producer-Consumer deployed for this experiment.

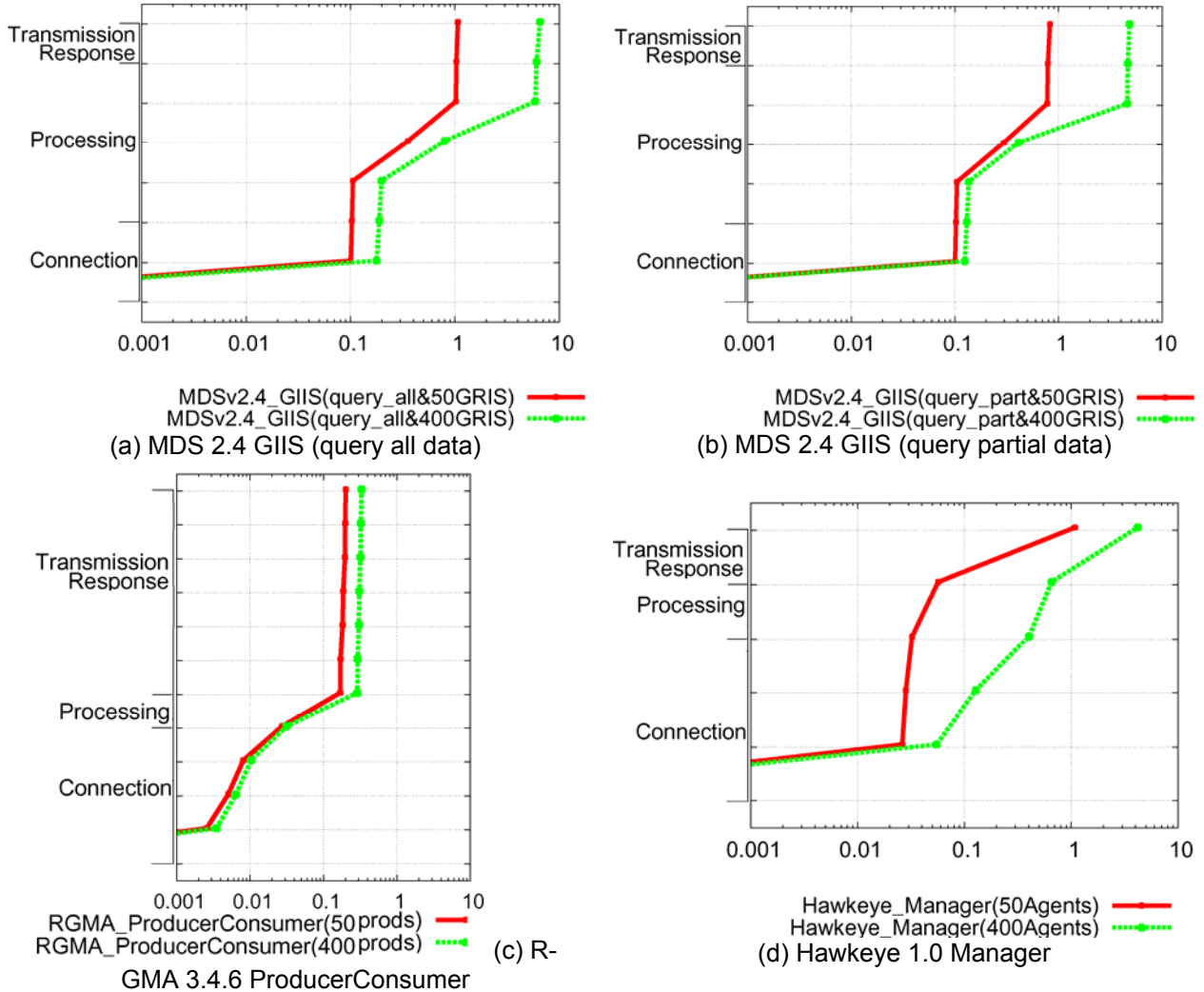


Figure 28: Aggregate Information Server performance by NetLogger step for 50 and 400 Information Collectors.

The Hawkeye Manager performs poorly in the response transmission and connection phases, but has the best performance when searching for ClassAds in the Processing phase. Much of the CORT is spent by the client receiving the query result from the server and processing the result, step 5 in Table 7. This is due to the constant increase in the query result size ranging from about 10 KB to 6 MB, directly proportional to the number of Agents. The machine hosting the Hawkeye Agent scaled well in terms of both load metrics.

3.7. Experimental Summary

Experiment 1 shows that caching can significantly improve the performance of the Information Server and is needed to allow the server to scale well with an increasing number of users. Deployments should carefully consider when a longer cache lifetime for increased performance may outweigh the disadvantages of an out-of-date element. In addition, when setting up an Information Server, care should be taken to make sure the server is on a well-connected machine, since the network behavior plays a larger role than expected for some implementations.

Experiment 2 also emphasizes the significance of network contention issues. The placement of a Directory Server on a well-connected machine plays a large role in determining scalability as the number of users grows. In addition, because significant loads are seen even with only a few users, it is important that this service run on a dedicated machine or that it be duplicated as the number of users grows.

Experiment 3 shows that too many Information Collectors can become the performance bottleneck for an Information Server, but we can use some general solutions to improve its performance. For example, we can let different sets of Information Collectors register to different Information Servers if needed, and then add in a layer to the hierarchy if needed.

Experiment 4 shows that, as currently implemented, none of the Aggregate Information Servers scaled well with the number of Information Servers registered to them. This strongly implies that when building hierarchies of aggregation, they will need to be rather narrow and deep, having very few Information Servers

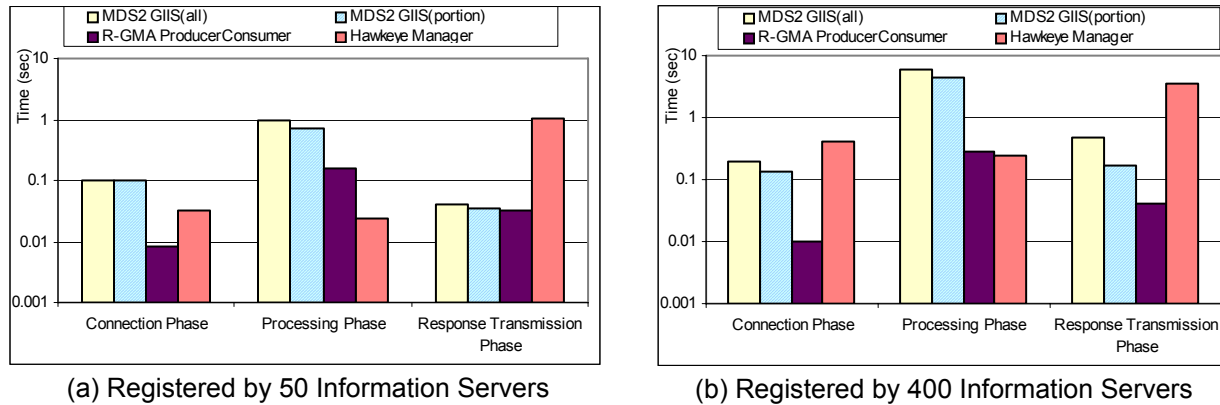


Figure 29: Aggregate Information server generic query phase comparison.

registered to any one Aggregate Information Server.

4. Deployment Recommendations

Given the experiments we have run, and due to the architectural and implementation approaches of the MIS systems we investigated, we can recommend several factors to consider when deploying the MIS systems. These are largely version-independent.

For an MDS deployment we recommend:

- The cache-time-to-live values should be set as large as feasible for data to improve the efficiency of the MDS2 GRIS and GIIS. Several experiments showed a significant improvement with caching.
- Each GRIS should be deployed such that no more than 100 users are expected to query it at a time or replicated to reduce the number of queries against it. Experiment 1, Figures 6 and 7, showed a significant throughput and CORT hit when there were more than 100 users querying a single GRIS. This was also seen in Experiment 2, Figure 12.
- A GRIS should manage less than 55 Information Providers. Experiment 3, Figure 20, shows that the load for a GRIS without caching is over 3 (considered overloaded) when only 20 IPs were registered, or 90 IPs when all data is cached. We recommend 55 IPs as a halfway point between these two and the assumption that at any time roughly half the data will be in cache.
- Fewer than 50 GRISes should register to a GIIS, as seen with the poor throughput performance in Experiment 4, Figure 24.

For an R-GMA deployment we recommend:

- A ProducerServlet should be duplicated so that no more than 50 Consumer queries occur at a time, as seen in Experiment 1, Figure 6. Replication of a ProducerServlet, however, requires duplicating the backend database, a significant technical challenge.

- Registries should be deployed so that no more than approximately 50 Consumers access it at a time. Experiment 2, Figure 12, shows a significant proportional drop in throughput at this point.
- More instances of ProducerServlet should be used to share a large number of Producers. Experiment 3 Figure 20 shows that the load is over 3 when there are 30 Producers.
- Very few StreamingProducers should register to any compound Producer-Consumer, as shown in Experiment 4, Figure 27.
- When deploying a ConsumerServlet, we also need to take into consideration how many Consumers will be querying it. Our experiments did not examine this deployment issue.

For a Hawkeye deployment we recommend:

- A Hawkeye Agent can support up to 300 concurrent queries, as shown in Experiment 1 Figure 7.
- The Manager can also support as many as 300 concurrent queries, as shown in Experiment 2, Figure 3, although it should use multiple threads for connections to help reduce user contention, an option that the Agent does not have.
- An Agent can support 90 modules, as shown by Experiment 3.
- The maximum number of Agents that should register to a Manager is approximately 200, as shown in Experiment 4.
- The user should add constraints to the query if possible to reduce the query result size and minimize the communication costs.

5. The Best of All Worlds: An Idealized MIS Implementation

Each of the approaches examined in our work had benefits and detractions. In an idealized implementation we could try to leverage many of the former and take the best aspects from each system in turn.

First and foremost, when designing an MIS implementation, performance and bottleneck analysis should be an aspect of the design, implementation and deployment from the first stages. Many of the performance limitations we found were inherent in basic design decisions made very early on, that had long reaching repercussions and no easy resolution. Likewise, it is important to know how the system will be used in order to tune it for that use. Depending on the number of expected users/queries, the amount of data to be held, and the type of query, very different design, implementation and deployment decisions should be made. As an example, if it were known a priori that there would be a very large number of users, we would concentrate on ways to reduce the connection time, as our results (especially Figures 10 and 16) show that as the number of users increase this is where the largest benefit could be found.

Improvements for an idealized MIS implementation during the connection or response phases fall into two broad categories, decreasing the amount of data or improving the network behavior. Decreased data can be achieved by using specialized formats (not something that adds significant padding, for example XML) or allowing partial results on queries so full data sets do not have to be transferred. Improved network behavior means just that- locating the servers in such a way that they are “close” to the clients – either by having better networks or replicating the servers in many places.

Processing time can also be improved in several ways. From our work it is obvious that caching is preeminent, so careful design work should allow for flexible, tunable caching, both active and passive as appropriate, in order to gain the benefits of a cache while lessening the effects of stale data whenever possible. We might also consider designing specialized indexes for common queries to improve response time, or having multiple indexing methods for a given data store to allow for better lookup performance. It should be noted that none of the MIS approaches we investigated discussed data models at all in their work, although it is well known in the database community that this will also play a large role in performance and should be considered strongly.

6. Related Work

Much work has been done on measuring the performance of networked, client/server systems that manage and serve information, similar to the three MIS systems we have examined here. The most closely related literature has examined directory service, Web servers, Web search engines, and database systems.

Directory services, such as DNS [38], map the names of network resources to their respective network addresses. Usually the data stored in a directory service is relatively static and the service employs caching to increase performance. Jung et al. [28] studied the effectiveness of DNS caching and found that the scalability of DNS was not as dependent on the hierarchical design of its name space or good A-record caching as was commonly believed, but that the ability to cache NS records and avoid overloading any single name server affected performance more directly. Their study was limited to measuring DNS interactions at only two points in the Internet topology. Liston et al. [34] identified various DNS performance metrics and studied location-related variations of these metrics from the perspective of the client. They concluded that the greatest performance enhancements could be achieved by reducing the response time of intermediate-level servers rather than the top-level root. Cohen et al. [7] proposed enhancements to passive DNS caching – renewal policies to refresh selected expired cached entries by issuing unsolicited queries and simultaneous-validation to transparently use expired records. These proactive caching strategies greatly reduced the fraction of HTTP connection establishments as well as the user-perceived latency.

Web server performance evaluation is also related to our work. Banga et al. [3] benchmarked a NCSA httpd Web server by modeling the effects of request overloads and showed it can sustain a maximum throughput around several hundred but less than 1,000 connections per second if running on fast uniprocessors. They also found the throughput decreases dramatically with the increase of average query response size. Their findings comply with the benchmarking results provided by WebStone [55]. This is similar to what we found for the MIS. Iyengar et al. [26] examined Web server performance in situations where server processing power is the limiting resource and found it is essential to cache dynamic pages to improve performance if the Web sites contain significant dynamic content. Habib et al. [22] analyzed the

latency in accessing Web pages by dividing the download time into four parts: DNS query, connection setup time, time to get the first byte of a Web page, and downloading time. They found the bottlenecks in accessing pages were the Internet itself and the 3-way TCP connection establishment, not the server speed. Several approaches have been proposed to improve the Web server performance, such as the Web server accelerators in [30] and distributed Web-server systems in [6, 31].

Caching also has a large effect on the performance of search engines. The experimental studies of AltaVista [2] in [49] and Excite [13] in [36, 56] show queries to search engines have significant locality and the majority of the repeated queries are referenced again within short time intervals. Google [4, 5], the most popular search engine today, employs the same technique to cache query results to improve search performance.

Besides DNS, Web servers, and search engines, the scalability of the prevalent database systems has been widely studied. Oracle 9i [16, 42] can achieve a throughput rating of 1,500 transactions per second and less than 1 second average response time when benchmarked with TPC-C benchmarks [54]. However, a transaction is not the same concept as the query in our performance study. A MySQL database [39] can support thousands of concurrent connections in theory; in practice, however, it can handle only a relatively small number of connections if the queries are memory and CPU intensive [59].

The performance study of an MIS or systems akin to an MIS used in distributed systems has much more recent origins. Smith et al. [51] investigated the Globus Toolkit® 1.1.3 MDS performance by focusing on the effect of different LDAP backends and data distribution strategies. Aloisio et al. [1] studied the capabilities and limitations of MDS2 as well as the security effect on the performance. However, their experiments were limited to simple tests on the GIIS only. Plale et al. [44, 45] benchmarked a set of queries and workload scenarios (synthetic workloads mimicking concurrent queries and updates from multiple users) against a nonrealistic Grid information service implemented with three platforms of different data models: relational, native XML, and LDAP; their study focused on a broad set of queries and scenarios issued against a rich data set. Keung et al. [32] analyzed the MDS2 GRIS performance with different back-end implementations

by varying information gathering methods. Their recent work [33] evaluated the effect of different implementations for the GRIS back-end on the performance of MDS2 GIIS. This work complements the MDS2 results discussed in our earlier paper [60]. Cooke et al. [9] repeated the scalability analysis of R-GMA by conducting the similar experiments as in our previous study [60] and their results are compatible with ours.

Not much work has been done to quantitatively compare the performance between different MIS systems. Plale et al. [43] discussed the pros and cons of building a Grid Information Service on a hierarchical representation and a relational representation. Two recent surveys [18, 58] presented a comprehensive comparison of a rich set of monitoring systems by classifying them using the authors' home-made taxonomies. However, all three were theoretical studies. Our previous work [60] examined the scalability of MDS2, R-GMA, and Hawkeye in a coarse-grained manner, focusing on the end-to-end measurements only. In a later work [61], we have examined the MDS2 behavior at a finer granularity by using NetLogger technologies to instrument the server and client codes, but we did not compare this behavior to any other system. A recent work from Schopf et al. [48] presented preliminary experimental results about MDS4's Index Service performance and showed it has a relatively better performance than the three MIS systems studied in our work.

7. Conclusions

In this paper, we evaluated the scalability of three MIS systems, MDS2, R-GMA, and Hawkeye, on a fine-grained level. NetLogger-assisted experiments were performed on each system to study their behaviors in more detail. Our work makes three contributions to the understanding of performance of an MIS implementation: it provides a quantitative report of the scalability for three prevalent MIS implementations; it reveals potential performance limitations by using NetLogger to better understand performance bottlenecks; and it offers several important suggestions to guide future deployments of MIS implementations, for example, a strong performance advantage to caching or prefetching the data, as well as

the need to have primary service components at well-connected sites because of the high load seen by all systems.

Acknowledgments

We thank John McGee and Ben Clifford at ISI for assistance with the MDS2; Andy Cooke and James Magowan for assistance with R-GMA; Alain Roy and Nick LeRoy at the University of Wisconsin, Madison, for assistance with Hawkeye; and Brian Tierney and Dan Gunter at LBNL for assistance with NetLogger. We also thank Scott Gose and Charles Bacon for assistance with the testbed at Argonne. This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under contract DE-AC02-06CH11357. .

Biographies

Xuehai Zhang received his B.S. degree in Computer Science from University of Science and Technology of China, Hefei in 1992. He received his M.S. degree in Computer Science from University of Chicago in 2002. He is currently a PhD candidate in the Department of Computer Science at University of Chicago. His research interests are in parallel and Grid computing, with emphasis on Grid systems' performance analysis and resource management on the virtual workspace, a virtualized Grid execution environment based on virtual machine technologies.

Jeffrey L. Freschl received his B.S. degree in Computer Science from the University of California, Santa Cruz in 2003. He received his M.S. degree in Computer Science from the University of Wisconsin, Madison in 2005. He is currently a Software Engineer with the DB2 Optimization group at IBM, Silicon Valley Lab. His research interests are in parallel and grid computing, with an emphasis in algorithm design and performance evaluation.

Jennifer M. Schopf is a Scientist at the Distributed Systems Lab, part of the Mathematics and Computer Science Division at Argonne National Lab, and is spending the year as a researcher at the National eScience Center in Edinburgh, UK. She is a member of the Globus Alliance, and technology coordinator for the MDS. She received a BA in Computer Science and Mathematics from Vassar College, and MS and PhD degrees from the University of California, San Diego in Computer Science and Engineering. Currently, her research interests include monitoring, performance prediction, and resource scheduling and selection.

References

- [1] G. Aloisio, M. Cafaro, I. Epicoco, and S. Fiore, "Analysis of the globus toolkit grid information service," GridLab, technical report GridLab-10-D.1-0001-GIS_Analysis, 2001.
- [2] AltaVista, <http://www.altavista.com/>.
- [3] G. Banga and P. Druschel, "Measuring the Capacity of a Web Server," Proceedings of USENIX Symposium on Internet Technologies and Systems, 1997.
- [4] L. Barroso, J. Dean, and U. Holzle, "Web Search for a Planet: The Google Cluster Architecture," in *IEEE Micro*, vol. 23, 2003.
- [5] S. Brin and L. Page, "The Anatomy of a Large-Scale Hypertextual Web Search Engine," *Computer Networks*, vol. 30(1-7), pp. 107-117, 1998.
- [6] V. Cardellini, M. Colajanni, and P. S. Yu, "Dynamic Load Balancing on Web-server Systems," in *IEEE Internet Computing*, vol. 3(3), 1999, pp. 28-39.
- [7] E. Cohen and H. Kaplan, "Proactive Caching of DNS Records: Addressing a Performance Bottleneck," Proceedings of 2001 Symposium on Applications and the Internet (SAINT 2001), 2001.
- [8] A. Cooke, A. Gray, L. Ma, W. Nutt, J. Magowan, P. Taylor, R. Byrom, L. Field, S. Hicks, and J. Leake, "R-GMA: An Information Integration System for Grid Monitoring," Proceedings of the 11th International Conference on Cooperative Information Systems, 2003.
- [9] A. Cooke, A. Gray, W. Nut, A. Cooke, A. Gray, W. Nutt, R. Cordenonsi, R. Byrom, L. Cornwall, A. Djaoui, Laurence Field, S. Fisher, S. Hicks, J. Leakey, R. Middleton, A. Wilson, X. Zhu, N. Podhorszki, B. Coghlan, S. Kenny, D. O'Callaghan, and J. Ryan, "The Relational Grid Monitoring Architecture: Mediating Information about the Grid," *Journal of Grid Computing*, vol. 2, 2004.
- [10] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman, "Grid information services for distributed resource sharing," Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing, 2001.
- [11] DataGrid WP3 Information and Monitoring Services, <http://hepunix.rl.ac.uk/edu/wp3>.
- [12] European DataGrid Project, <http://www.eu-datagrid.org>.
- [13] Excite, <http://www.excite.com>.
- [14] S. Fisher, "Relational model for information and monitoring," GGF, Technical report GWD-Perf-7-1, 2001 2001.
- [15] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke, "Security architecture for computational grids," Proceedings of the 5th ACM Conference on Computer and Communications Security (CCS-5), 1998.
- [16] "Full Disclosure Report for HP ProLiant DL580 - PDC 32P", Hewlett Hewlett-Packard Company, 3rd ed., 2002.
- [17] Ganglia, <http://ganglia.sourceforge.net>.
- [18] M. Gerndt, R. Wismueller, Z. Balaton, G. Gombas, P. Kacsuk, Z. Nemeth, N. Podhorszki, H.-L. Truong, T. Fahringer, M. Bubak, E. Laure, and T. Margalef, "Performance Tools for the Grid: State of the Art and Future," Lehrstuhl fuer Rechnertechnik und Rechnerorganisation (LRR-TUM) Technische Universitaet Muenchen. Vol. 30 of Research Report Series, 2004.
- [19] Global Grid Forum, <http://www.gridforum.org>.
- [20] Globus Alliance, <http://www.globus.org>.
- [21] D. Gunter and B. Tierney, "NetLogger: A Toolkit for Distributed System Performance Tuning and Debugging," Proceedings of Integrated Network Management 2003, 2003.
- [22] A. Habib and M. Abrams, "Analysis of Sources of Latency in Downloading Web Pages," Proceedings of Webnet, 2000.
- [23] Hawkeye, <http://www.cs.wisc.edu/condor/hawkeye>.
- [24] Inca, <http://tech.teragrid.org/inca/>.
- [25] Iperf, <http://dast.nlanr.net/Projects/Iperf>.

- [26] A. Iyengar, E. MacNair, and T. Nguyen, "An Analysis of Web Server Performance," Proceedings of IEEE Global Internet 1997, 1997.
- [27] Java Servlet Technology, <http://java.sun.com/products/servlet>.
- [28] J. Jung, E. Sit, H. Balakrishnan, and R. Morris, "DNS performance and the effectiveness of caching," Proceedings of ACM SIGCOMM Internet Measurement Workshop, 2001.
- [29] M. Knop, J. Schopf, and P. Dinda, "Windows Performance Monitoring and Data Reduction using WatchTower," Proceedings of SHAMAN, 2002.
- [30] E. Levy-Abegnoli, A. Iyengar, J. Song, and D. Dias, "Design and Performance of a Web Server Accelerator," Proceedings of INFOCOM, 1999.
- [31] Q. Li and B. Moon, "Distributed Cooperative Apache Web Server," Proceedings of 10th International World Wide Web Conference, 2001.
- [32] H. N. Lim Choi Keung, J. R. D. Dyson, S. A. Jarvis, and G. R. Nudd, "Performance evaluation of a grid resource monitoring and discovery service," *IEEE Proceedings: Software*, vol. 150, pp. 243-251, 2003.
- [33] H. N. Lim Choi Keung, J. R. D. Dyson, S. A. Jarvis, and G. R. Nudd, "Predicting the Performance of Globus Monitoring and Discovery Service (MDS-2) Queries," Proceedings of the 4th International Workshop on Grid Computing, 2003.
- [34] R. Liston, S. Srinivasan, and E. W. Zegura, "Diversity in DNS Performance Measures," Proceedings of Internet Measurement Workshop (IMW), 2002.
- [35] M. J. Litzkow, M. Livny, and M. W. Mutka, "Condor - a hunter of idle workstations," Proceedings of the 8th International Conference on Distributed Computing Systems, 1988.
- [36] E. P. Markatos, "On Caching Search Engine Results," in *Computer Communications*, vol. 24(2001), 2001, pp. 137-143.
- [37] MDS2, <http://www.globus.org/mds/mds2>.
- [38] P. Mockapetris, "Domain names - Concepts and Facilities," IETF RFC1034, 1987.
- [39] MySQL, <http://www.mysql.com/>.
- [40] Network Time Protocol (NTP), <http://www.ntp.org/>.
- [41] OpenLDAP, <http://www.openldap.org>.
- [42] Oracle, <http://www.oracle.com>.
- [43] B. Plale, P. Dinda, and G. v. Laszewski, "Key concepts and services of a grid information service," Proceedings of the 15th International Conference on Parallel and Distributed Computing Systems (PDCS), 2002.
- [44] B. Plale, C. Jacobs, S. Jensen, Y. Liu, C. Moad, R. Parab, and P. Vaidya, "Understanding Grid Resource Information Management through a Synthetic Database Benchmark/Workload," Proceedings of the 4th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid2004) (to appear), 2004.
- [45] B. Plale, C. Jacobs, Y. Liu, C. Moad, R. Parab, and P. Vaidya, "Benchmark Details of Synthetic Database Benchmark/Workload for Grid Resource Information," Indiana University Computer Science Technical Report TR-583 Technical Report TR-583, August 2003 2003.
- [46] R. Raman, "Matchmaking Frameworks for Distributed Resource Management," University of Wisconsin, PhD thesis 2000.
- [47] R. Raman, M. Livny, and M. Solomon, "Matchmaking: Distributed resource management for high throughput computing," Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing, 1998.
- [48] J. M. Schopf, M. D'Arcy, N. Miller, L. Pearlman, I. Foster, and C. Kesselman, "Monitoring and Discovery in a Web Services Framework: Functionality and Performance of the Globus Toolkit's MDS4," Argonne National Laboratory Technical Report #ANL/MCS-P1248-0405, 2004.
- [49] C. Silverstein, M. Henzinger, and H. Marais, "Analysis of a very large altavista query log," Digital SRC, Technical report #1998-014, 1998.
- [50] S. Smallen, C. Olschanowsky, K. Ericson, B. P., and J. M. Schopf, "The Inca Test Harness and Reporting Framework," Proceedings of the 2004 ACM/IEEE conference on Supercomputing, 2004.

- [51] W. Smith, A. Waheed, D. Meyers, and J. Yan, "An Evaluation of alternative designs for a grid information service," Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing (HPDC-9), 2000.
- [52] B. Tierney, R. AYDT, D. GUNTER, W. SMITH, V. TAYLOR, R. WOLSKI, and M. SWANY, "A grid monitoring architecture," GGF, technical report GWD-PERF-16-2, January 2002 2002.
- [53] B. Tierney, W. Johnston, B. Crowley, G. Hoo, C. Brooks, and D. Gunter, "The NetLogger Methodology for High Performance Distributed Systems Performance Analysis," Proceedings of IEEE High Performance Distributed Computing conference, 1998.
- [54] Transaction Processing Performance Council (TPC), <http://www.tpc.org>.
- [55] WebStone, <http://www.mindcraft.com/webstone/>.
- [56] Y. Xie and D. O'Hallaron, "Locality in Search Engine Queries and Its Implications for Caching," Proceedings of INFOCOM 2002, 2002.
- [57] W. Yeong, T. Howes, and S. Kille, "Lightweight Directory Access Protocol," IETF RFC1777, 1995.
- [58] S. Zaniolas and R. Sakellariou, "A Taxonomy of Grid Monitoring Systems," in *Future Generation Computing Systems*, vol. 21, 2005, pp. 163-188.
- [59] J. Zawodny and D. J. Balling, "Server Performance Tuning," in *High Performance MySQL*, 1st ed: O'Reilly, pp. 102-129, 2004.
- [60] X. Zhang, J. Freschl, and J. M. Schopf, "A performance study of monitoring and information services for distributed systems," Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC-12), 2003.
- [61] X. Zhang and J. M. Schopf, "Performance Analysis of the Globus Toolkit Monitoring and Discovery Service, MDS2," Proceedings of IEEE IPCCC International Workshop on Middleware Performance (IWMP 2004), 2004.